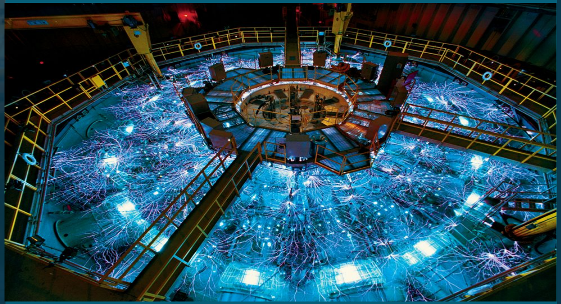
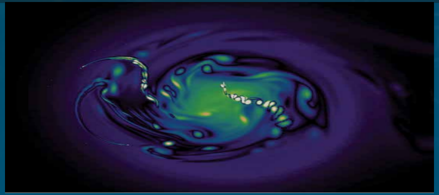
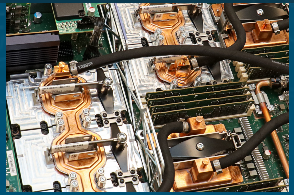
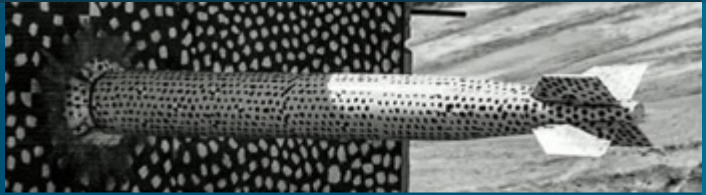
Introduction to the Structural Simulation Toolkit (SST)









PRESENTED BY

Si Hammond and Clay Hughes, Scalable Computer Architectures

SAND2018-3453 PE

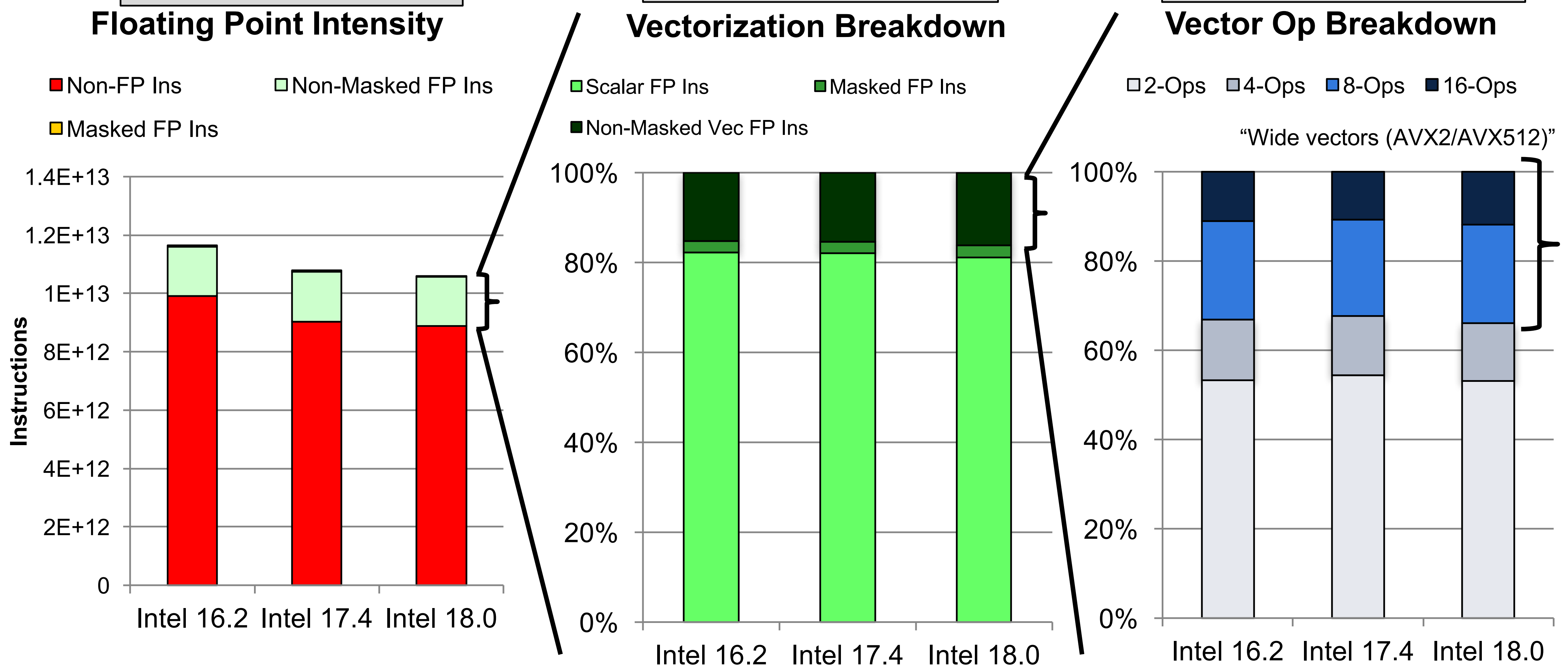
04/03/2018

Overview of What our Applications Really Look Like…

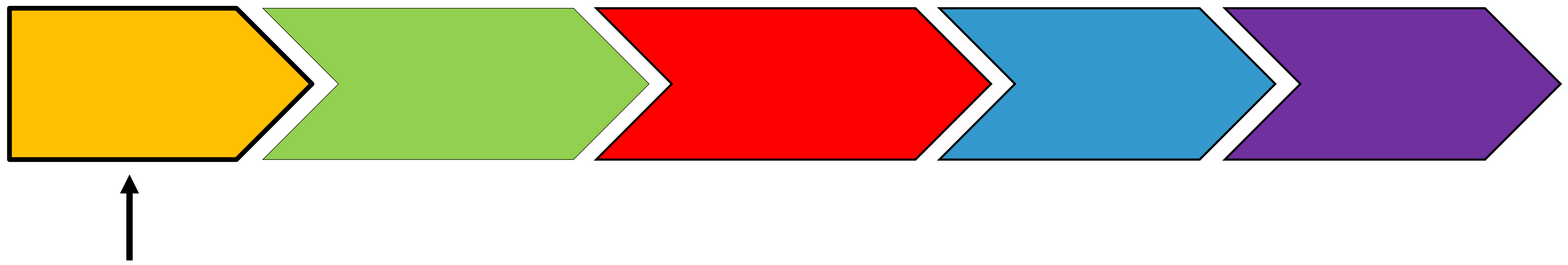
Approx 15% is FP Instruction

Approx 20% of FP is vectorized

Approx 35% is "wide" vectorized



The Path to the Hardware of Tomorrow



One Vendor, One Solution (which tries to satisfy everyone)

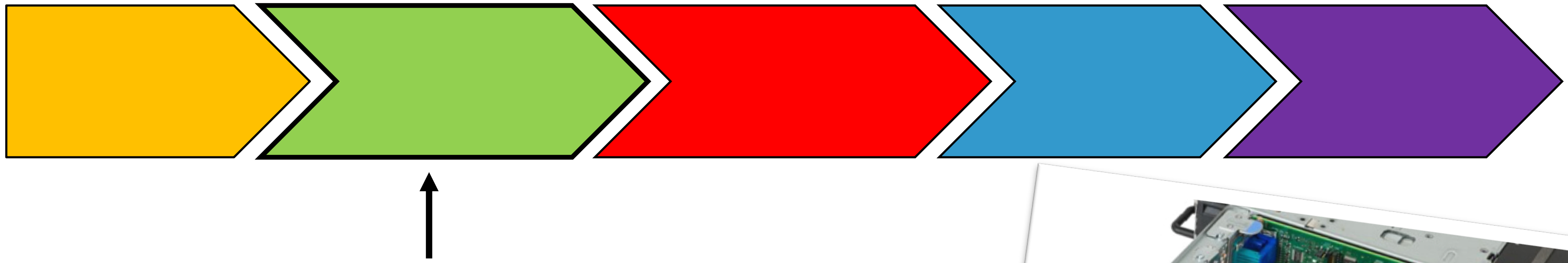
One vendor integrates everything, one fixed solution for every problem

Probably in the rearview mirror for most of HPC systems now

Little flexibility, less hardware tuning

Take what you get given (usually get lots you don’t want)

The Path to the Hardware of Tomorrow



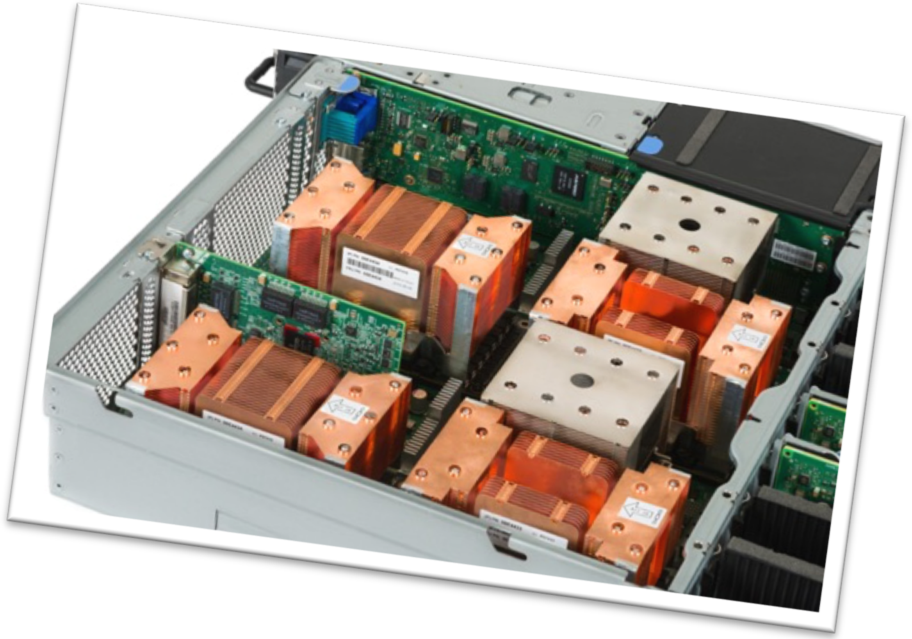
Multi Vendor, Protocol Integration

Multiple vendors working together, not integrated hardware

Support for a shared protocol

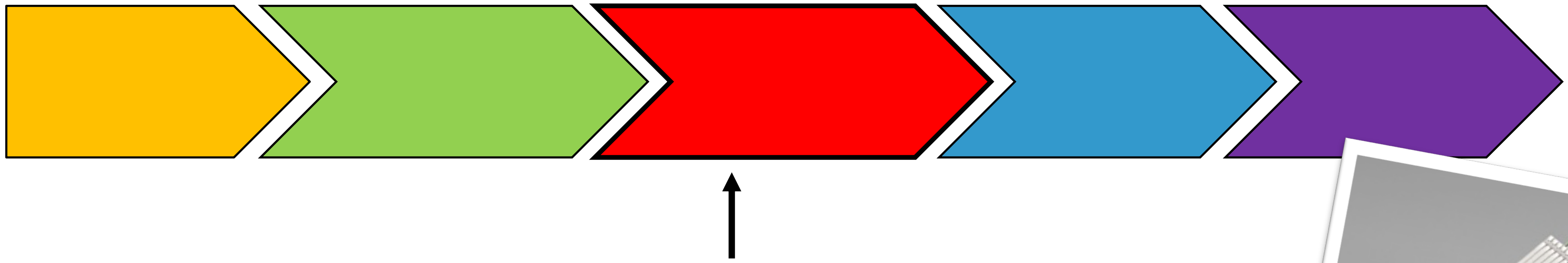
Seeing this with emerging IBM + NVIDIA solutions

CCIX, Gen-Z, OpenCAPI *etc*



IBM NVIDIA
Source: NVIDIA

The Path to the Hardware of Tomorrow



One Vendor, “Shopping List SoC”

Buy a solution from one vendor from an IP catalogue

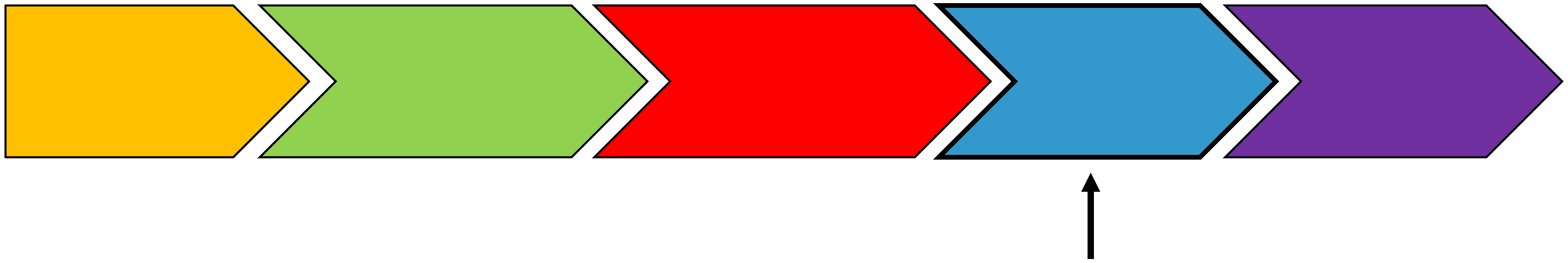
Cherry pick components which are optimized/appropriate for your workload

Complex task for mixed workload environments like DOE (not easy)

Does one vendor have everything you need?

Google TPU
Source: Google

The Path to the Hardware of Tomorrow



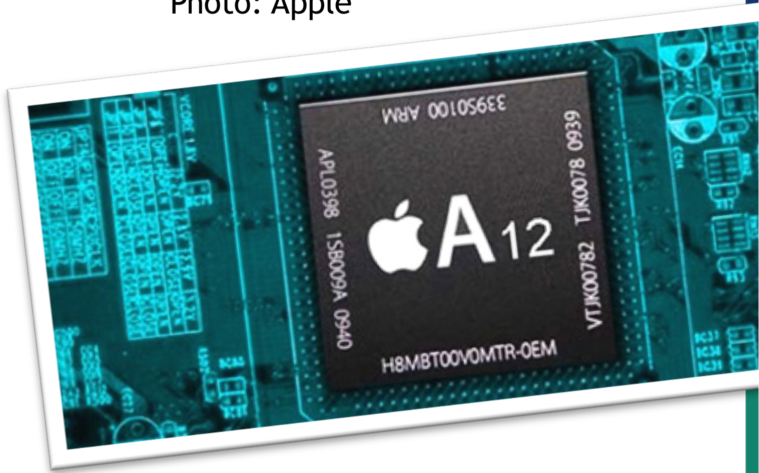
Customer + Silicon Provider SoC

Custom IP included in SoC

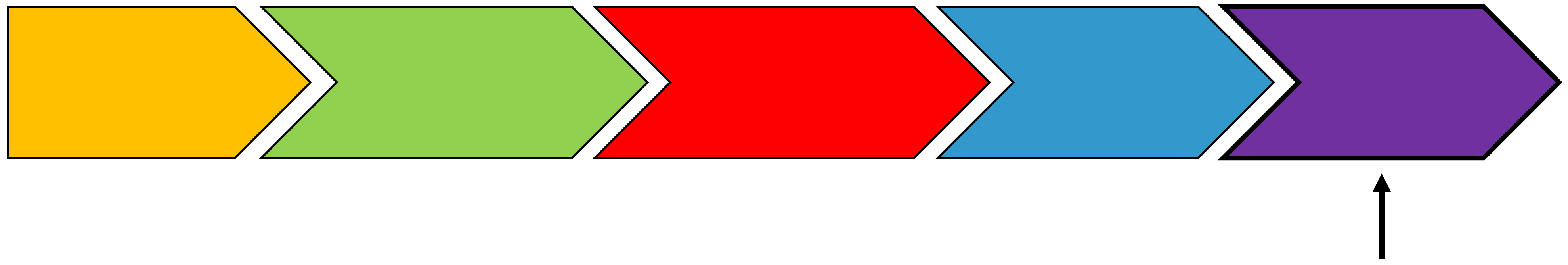
Becomes true plug and play for hardware, allows customization where workload permits

Issue is validation. Open source hardware/HPC has a role to play

Apple A12 SoC
Photo: Apple



The Path to the Hardware of Tomorrow



Conventional IP Library + Non-Conventional IP in SoC

Emergence of non-conventional IP (quantum, neuromorphic, *etc*) in a single SoC

Possible from IP catalogue but could be custom/open source components

Potentially greater optimization across complex workloads

Thoughts on the Long Term Path

Exciting to see huge potential far in the future for HPC

- The best of HPC really is yet to come, this is just the beginning

But .. this is a really complex path to follow

Needs a <u>very</u> good understanding of workloads, not just hardware

Diverse and flexible IP catalogue (commercial and open source)

Validation is the challenge (particularly for *real* solutions in HPC space)

Significant upside for some workloads but not all

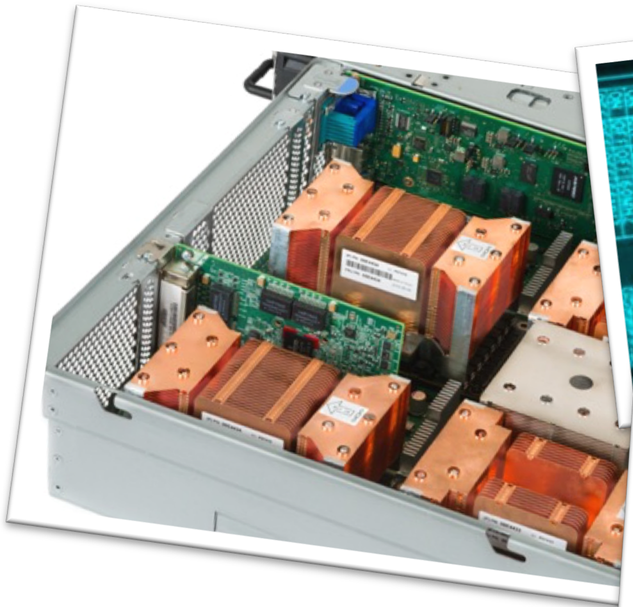
Gives life into silicon even when Moore's law finally is dead!

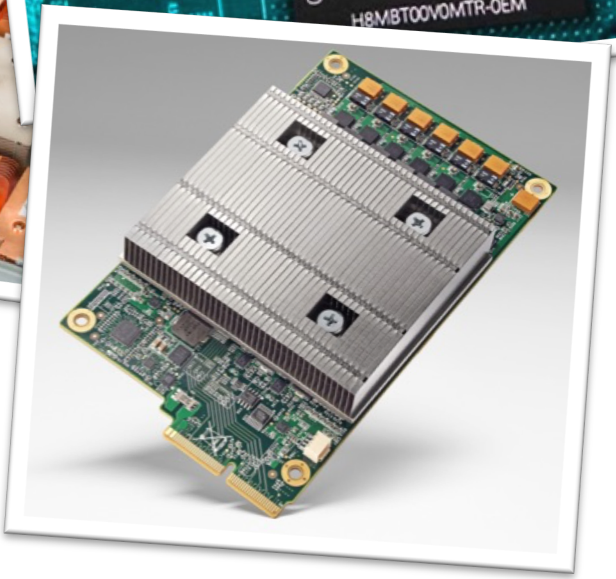
But You're Supposed to be Talking about SST…

In order to realize such a diverse hardware-jungle future, we need a really flexible, agile way to simulate/emulate potential future systems

In an ideal world:

- Very fast to execute simulator
- Broad cross-section models
- Models that play well together
- Extensible as new ideas emerge





Overview of SST

Why SST?

Problem: Simulation is slow

- Tradeoff between accuracy and time to simulate
- Many simulators are serial, unable to simulate very large systems

Problem: Lack of simulator flexibility

- Tightly-coupled simulations: Difficult to modify
- Difficult to simulate at different levels of accuracy

The Structural Simulation Toolkit:
A parallel, discrete-event simulation framework
for scalability and flexibility

What is SST?

Goals
- Become the standard architectural simulation framework for HPC
- Be able to evaluate future systems on DOE/DOD workloads
- Use supercomputers to design supercomputers

Status
- Parallel Core, basic components
- Current Release (7.2)
  - Improved components
  - Modular core/elements
  - More Internal documentation

Technical Approach
- Parallel
  - Parallel Discrete Event core with <u>conservative optimization over MPI/Threads</u>
- Multiscale
  - Detailed and simple models for processor, network, & memory
- Interoperability
  - DRAMSim, ,memory models
  - routers, NICs, schedulers
- Open
  - Open Core, non-viral, modular

Consortium
- "Best of Breed" simulation suite
- Combine Lab, Academic & Industry

Key Capabilities

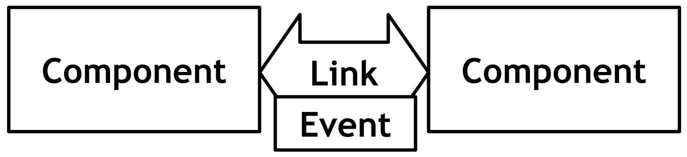
Parallel

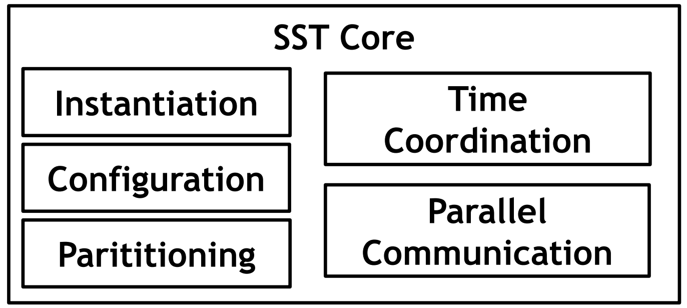
- Built from the ground up to be scalable
- Demonstrated scaling to 512+ host processors
- Conservative, Distance-based Optimization
- MPI + Threads

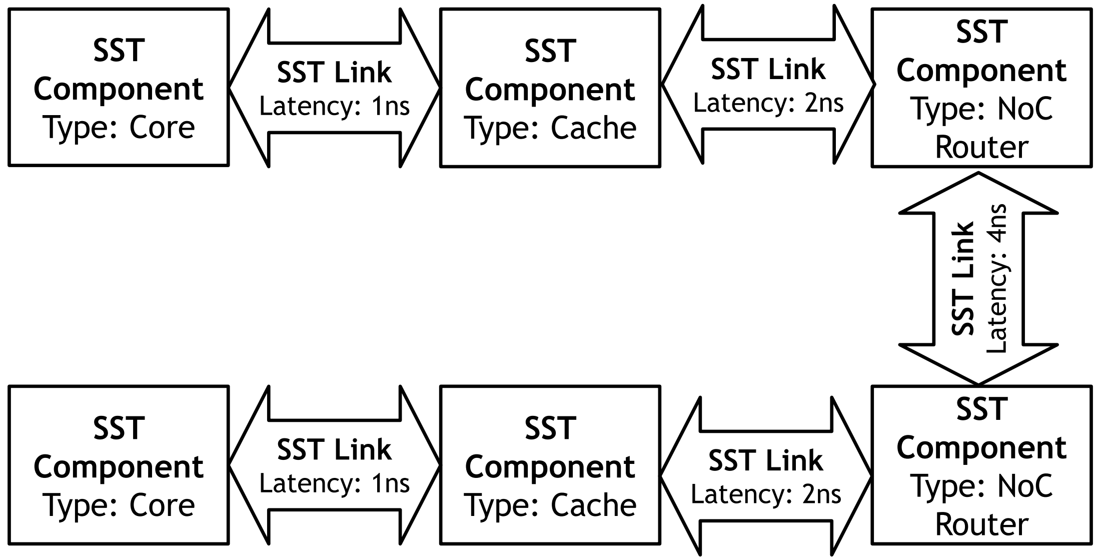
Flexible

- Enables "mix and match" of simulation components
- Multiscale tradeoff between accuracy and simulation time
  - *e.g.*, cycle-accurate network with trace-driven endpoints
- Open API
  - Easily extensible with new models
  - Modular framework
  - Open-source core (but BSD licensed so works with commercial partners too)

SST's Discrete-Event Algorithm







Simulations are comprised of **components** connected by **links**

Components interact by sending events over **links**

Each **link** has a minimum latency

Components can load **SubComponents** and **modules** for additional functionality

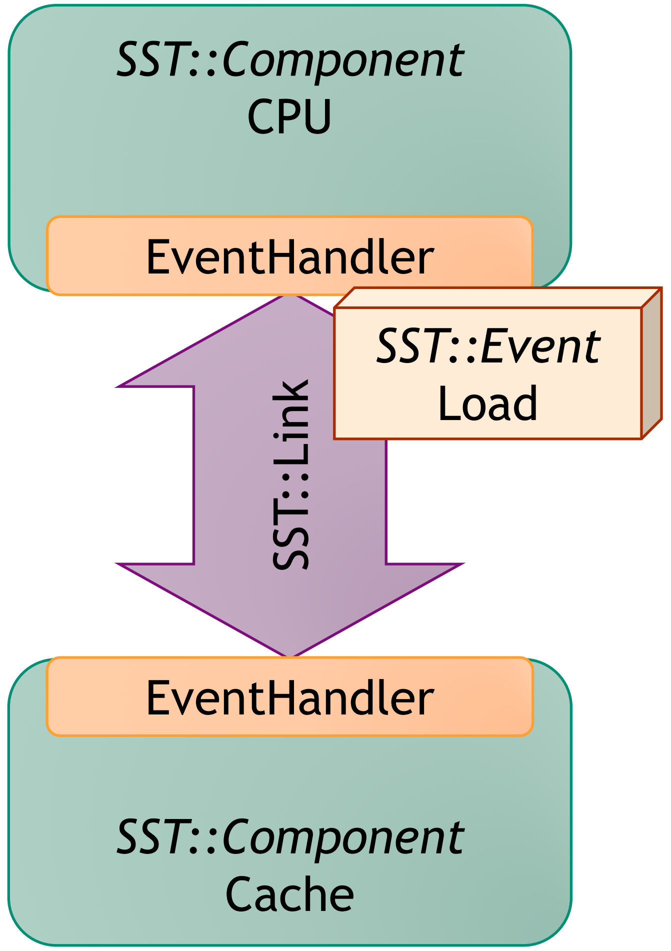
Key Simulation Objects

SST::Component
- Simulation model

SST::Link
- Communication path between two components
- Has optional EventHandler

SST::Event
- A discrete event

SST::Clock::Handler
- Function to handle a clock tick



Component

Basic building block of a simulation model → Performs actual simulation

- *e.g.*, processor, cache, network router, etc.

Uses Links and Ports to communication with other components

- Components define ports, links connect ports between components
- Polled: Register a clock handler to poll the link
- Interrupt: Register an event handler to be called when an event arrives
- Both: Receive events on interrupt, send events on clock

Link

Connects two components

- Connect a specific "Port" on component A to a "Port" on component B

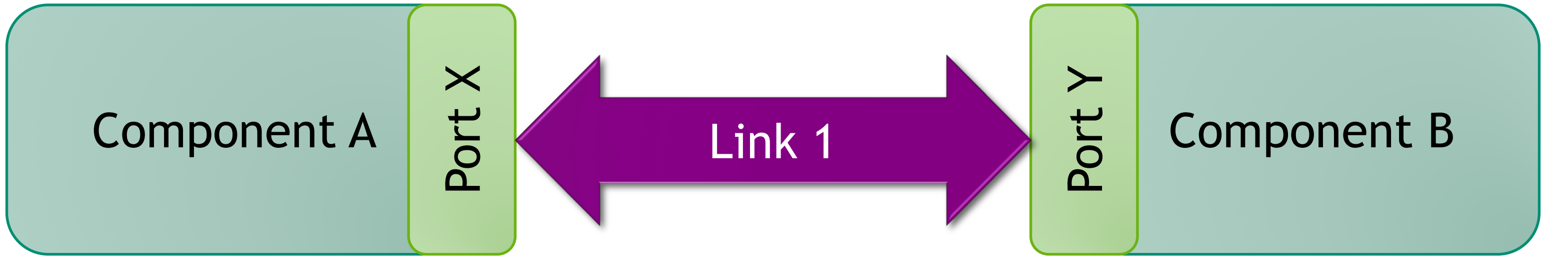
The ONLY mechanism by which components communicate

- Necessary for parallel simulation

Has a minimum, non-zero latency for communication

- Except self-links
- Except during initialization (untimed)

Transparently handles any MPI / threaded communication



Event

Unit of communication between two components

- Packet format is up to the communicating components

Some standardized interfaces

- Facilitate "mix and match" capability
- sst/core/interfaces/
- Memory (simpleMem)
  - Defines commands & event format for communication with memory
- Network (simpleNetwork)
  - Defines a header for events sent through a network component

Simulation lifecycle

Birth

- Create graph of components using Python configuration file
- Partition graph and assign components to MPI ranks
- Instantiate components and connect via links
- Initialize components using their init() functions
- Setup components using their setup() functions

Life

- Send events
- Manage clock and event handlers

Death

- Finalize components using their finish() functions
- Output statistics
- Cleanup simulation, delete components

Example Studies

IBM Memory Controller Design

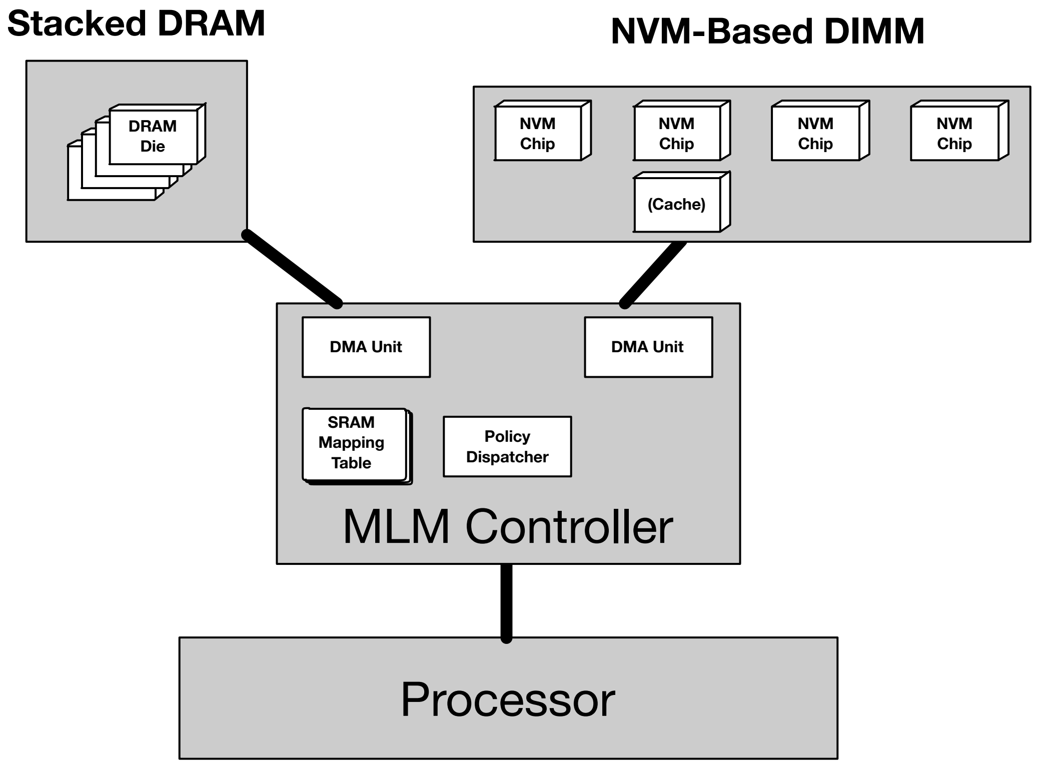
Improvements to IBM CramSim to enable threaded simulations (faster analysis time)

Improved multi-level memory models

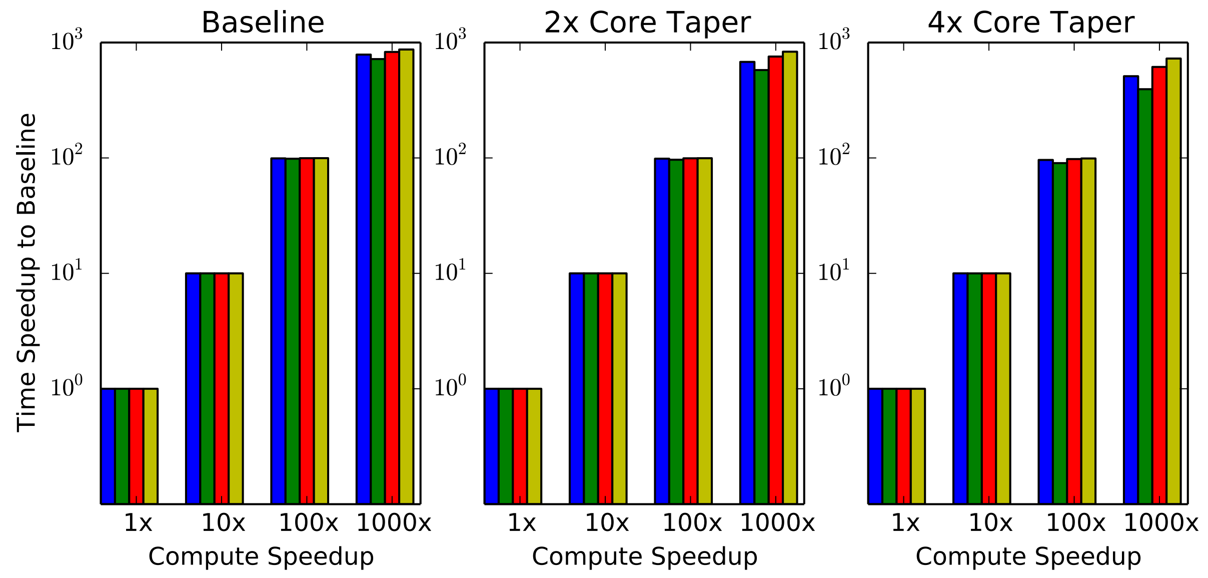
Performance and scaling improvements (event-driven (clock-less) memory models)

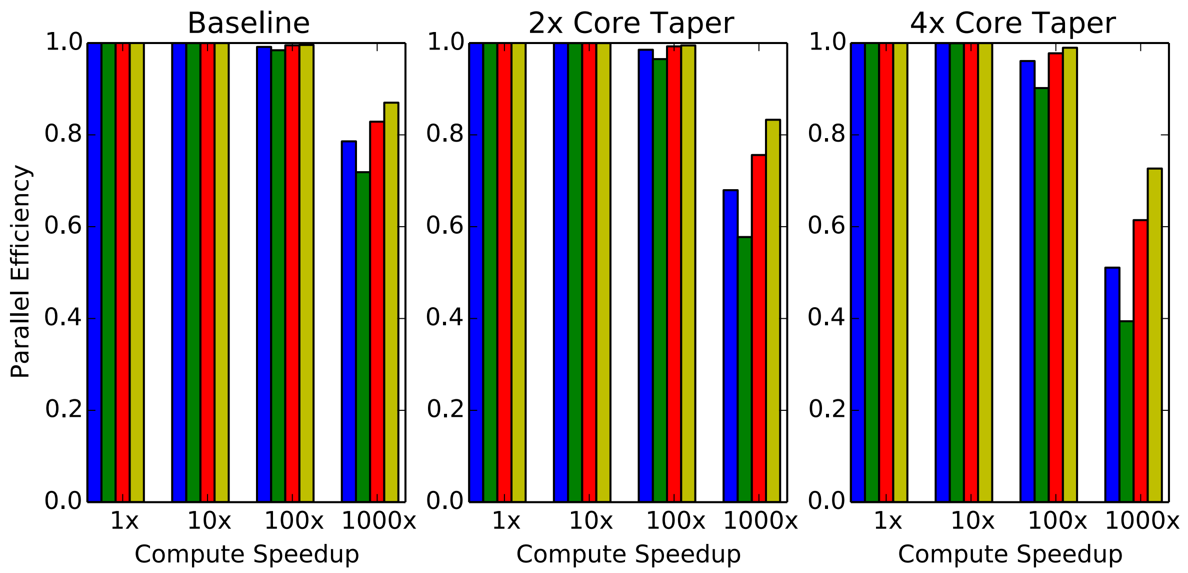
Scratchpad support

New TLB model



Network Bandwidth Tapering





Questions that most frequently come up in DOE procurements

Unstructured nearest neighbor study shows little sensitivity to tapering (save money)

🟦	Linear
🟩	Random
🟥	TreeMatch
🟨	APHiD

Multi-Level Memory/PIM Analysis

Processing-in-memory

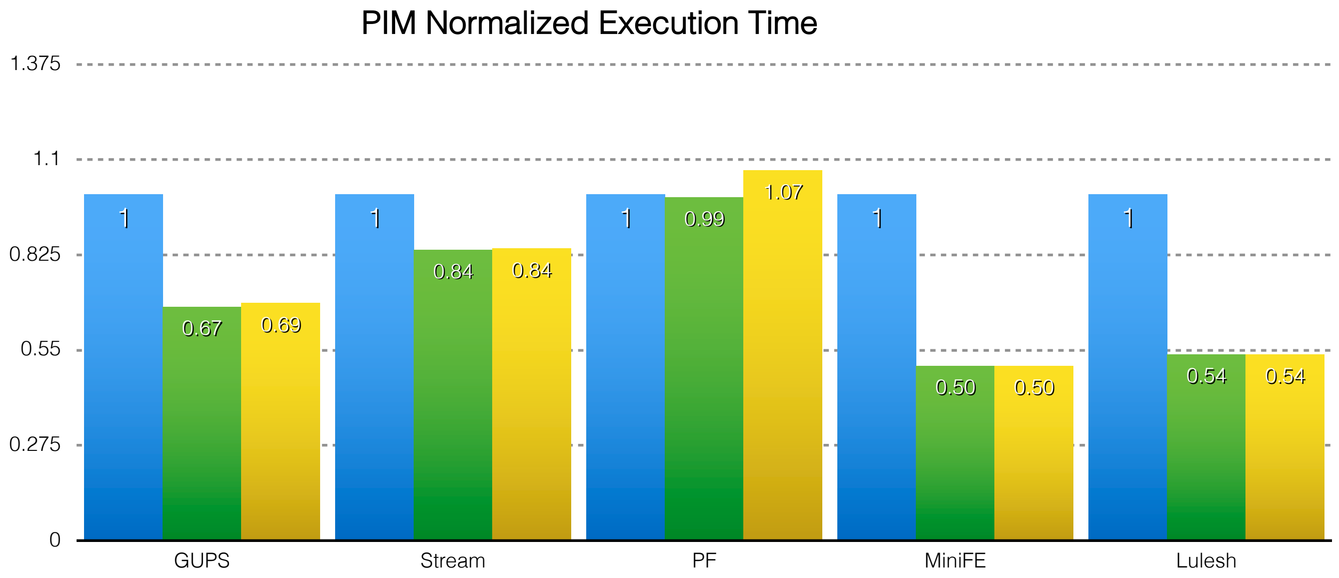
Multi-Level Memory

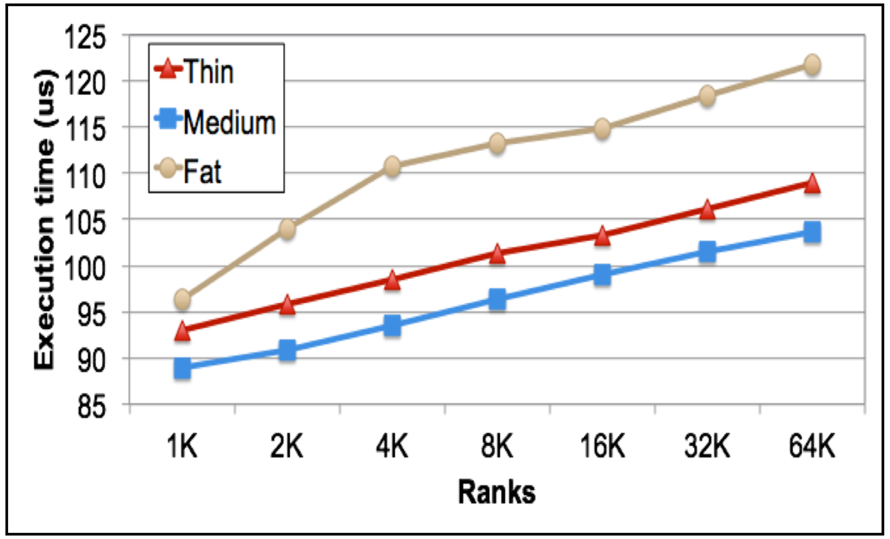
- HW Tradeoffs: capacity ratios,
- SW Tradeoffs: application, runtime, OS, HW control

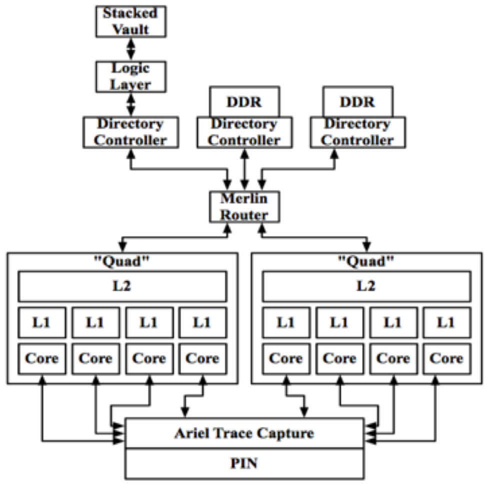
Scalable Network Studies

- Network on Chip
- Cache coherency

Scheduling







Deep(er) Dive on SST

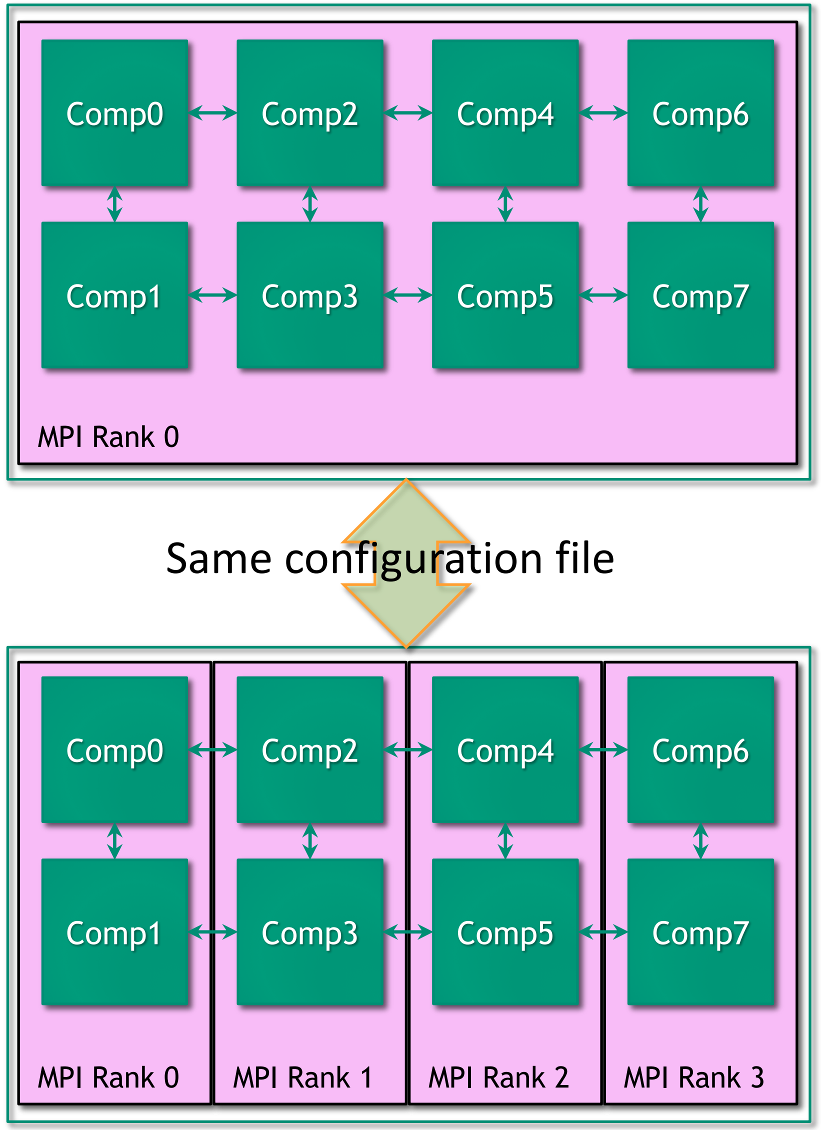
SST in Parallel

SST was designed from the ground up to enable scalable, parallel simulations

Components are distributed among MPI ranks & threads

Links allow parallelism

- Hence, components should communicate via links only
- Transparently handle any MPI communication
- Specified link-latency determines MPI synchronization rate

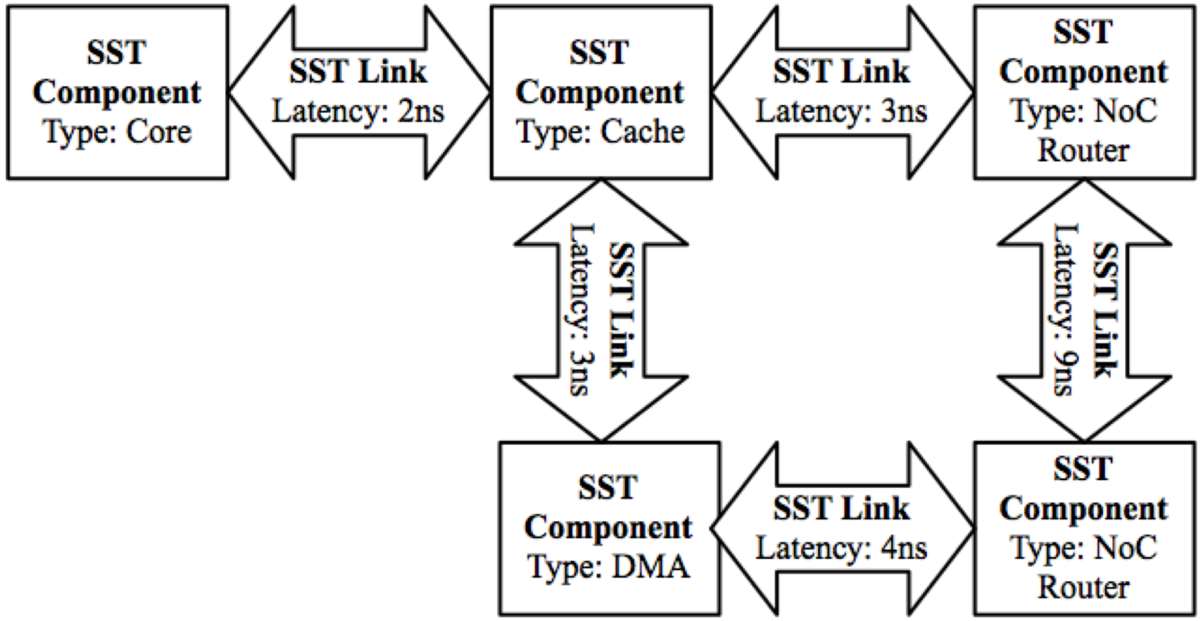


SST Discrete Event Algorithm

SST Simulation are comprised of components connected by links

Components interact by ending events over links

Each link has a minimum latency (specified in SI time units)



Existing SST Element Libraries

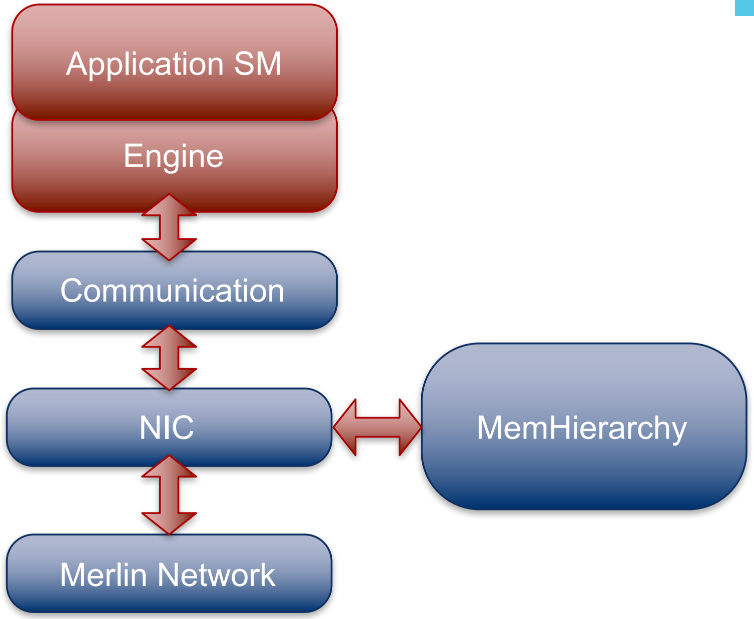
Detailed Memory Models	memHierarchy	- Cache and Memory
	cassini	- Cache prefetchers
	DRAMSim	- DDR
	NVDIMMSim	- Emerging Memories
	Goblin	– HMC
Dynamic Trace-based Processor Model	ariel	- PIN-based Tracing
Cycle-based Processor Model	m5C	- Gem5 integration layer
High-level Program Communication Models	ember	- State-machine Message generation
	firefly	- Communication Protocols
	hermes	- MPI-like interface
Cycle-based Network Model	Merlin	- Network router model and NIC
High-level System Workflow Model	scheduler	- Job-scheduler simulation models

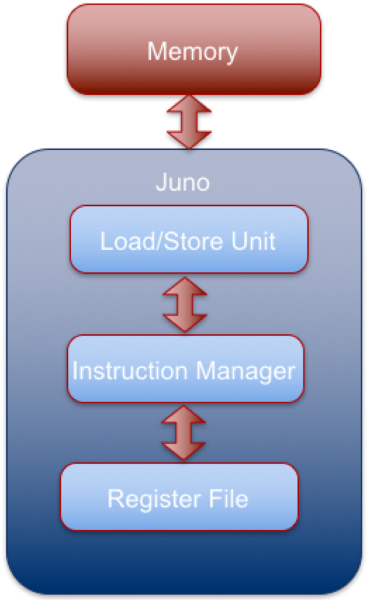
New Additions to SST: Core

1. Overhaul of the SST element information description system
2. Improved external component support (much easier to support than external development community)
3. HDF5 & JSON support for statistics output
4. Improved thread scaling
5. Easier Builds (removed Boost Dependency)
6. Early support for running SST on IBM POWER
7. Implementation of a “stop at” wall time feature for working within scheduled cluster environments
8. Support for describing co-ordinates of components in Python configuration (for visualizations)

New Additions to SST

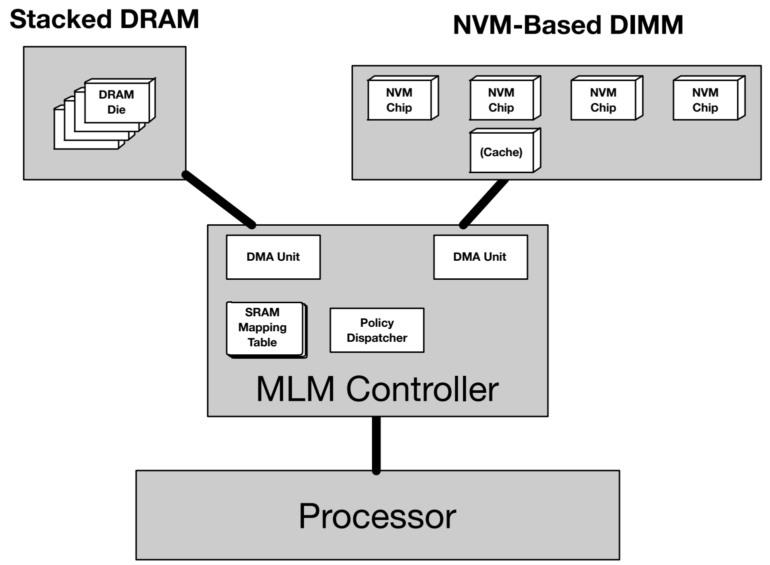
1. Support for memory modeling in large-scale network analysis
2. Early support for some SHMEM based communication models (in progress)
3. Juno Processor Model
   - Simplified processor model
   - Designed for extensibility
   - Uses: Tutorial, Correctness checking





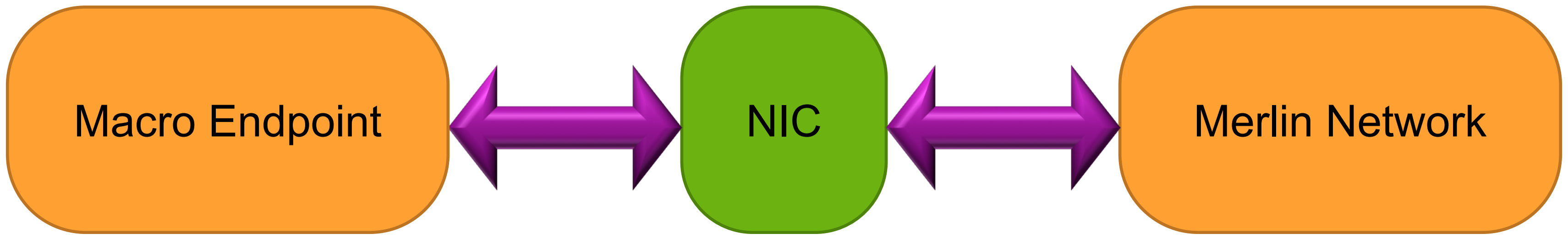
New Additions : Memory

1. Improvements to IBM CramSim for enabling threaded simulations
2. Improved multi-level memory models
3. Performance and scaling improvements (event-driven (clock-less) memory models)
4. Scratchpad support
5. New TLB model



New Additions : SST/Macro

1. Macro / Merlin Integration
2. Beta release of OTF2 trace replay skeleton
3. Beta release of Clang-based auto-skeletonization source-to-source tools
4. Integrated job launcher components for simulating PBS or SLURM-like batch systems

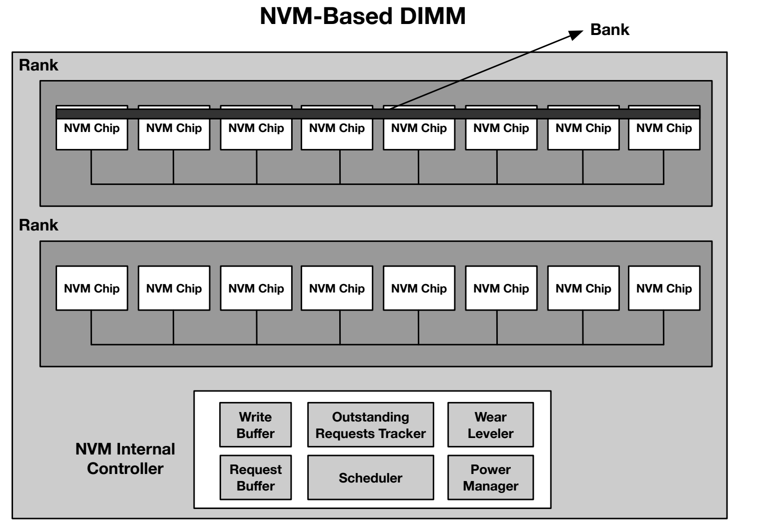


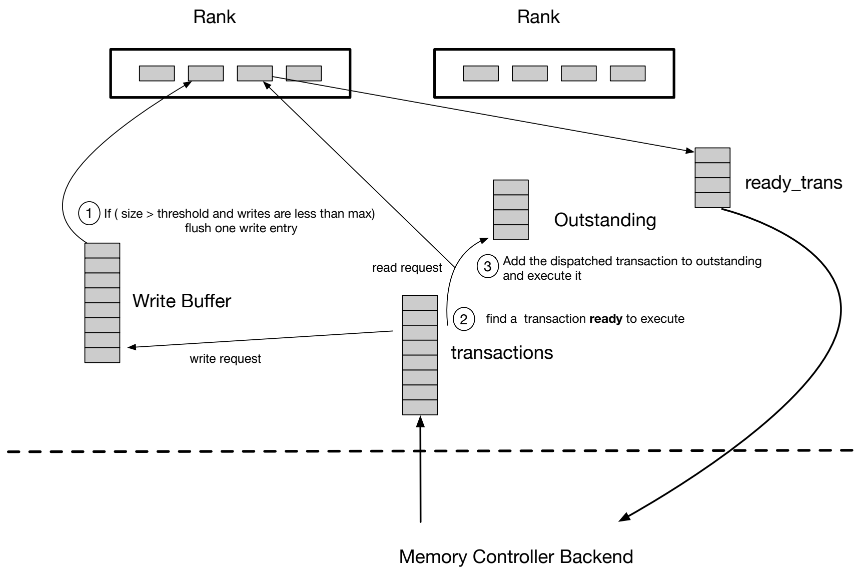
New Additions : Components : Messier

Messier: NV Memory model

Focus on NV-DIMMs e.g.:

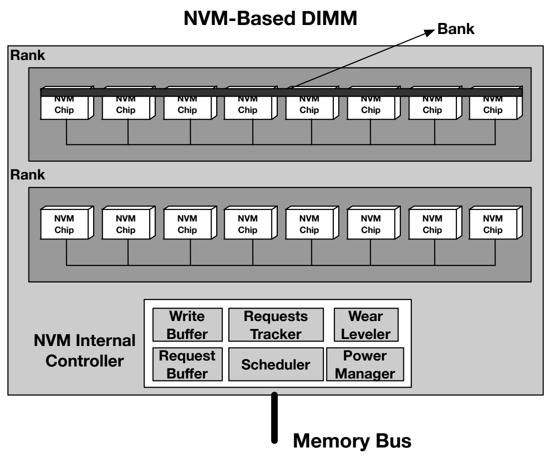
- # Banks, Latencies
- Row buffers, write buffers
- policies, outstanding requests, ordering
- Address mapping



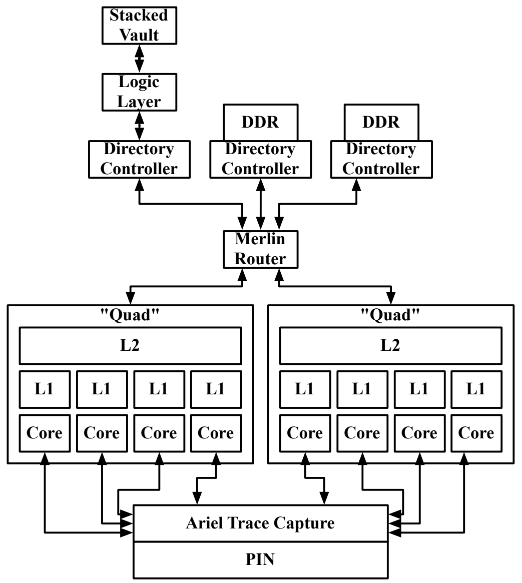

Rank
Rank
ready_trans
1 If (size > threshold and writes are less than max) flush one write entry
Outstanding
read request
3 Add the dispatched transaction to outstanding and execute it
Write Buffer
2 find a transaction ready to execute
transactions
write request
Memory Controller Backend

Report: SAND2017-1830

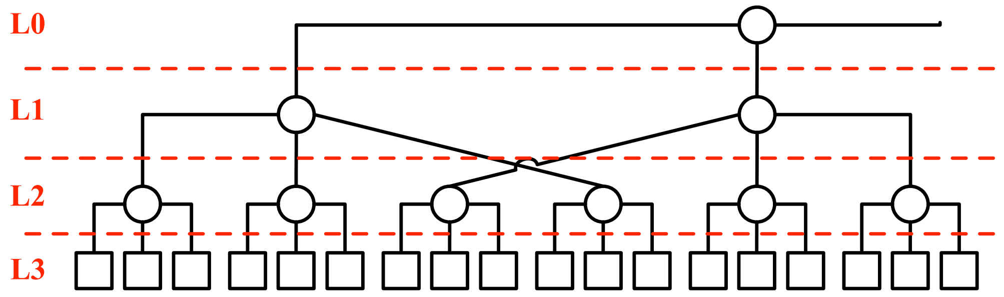
More SST Use Cases



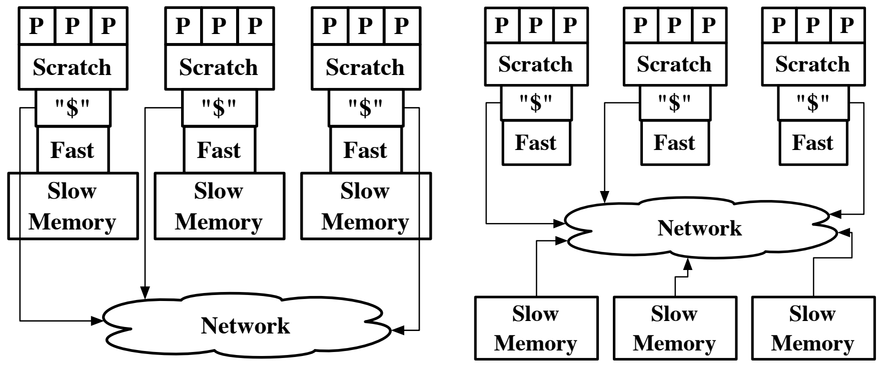
Emerging NVM Technologies



Multi-Level Memory (HBM+DDR+NV)



Photonic Network Topology & Routing



Dissaggregated Memory

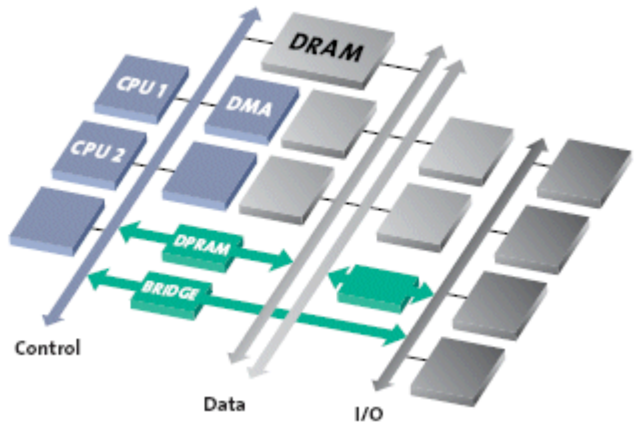
Future Directions

HDL Simulation via Verilator & Chisel

- Low-level hardware design
- Path to tape-out (Chisel)

New Processor Models

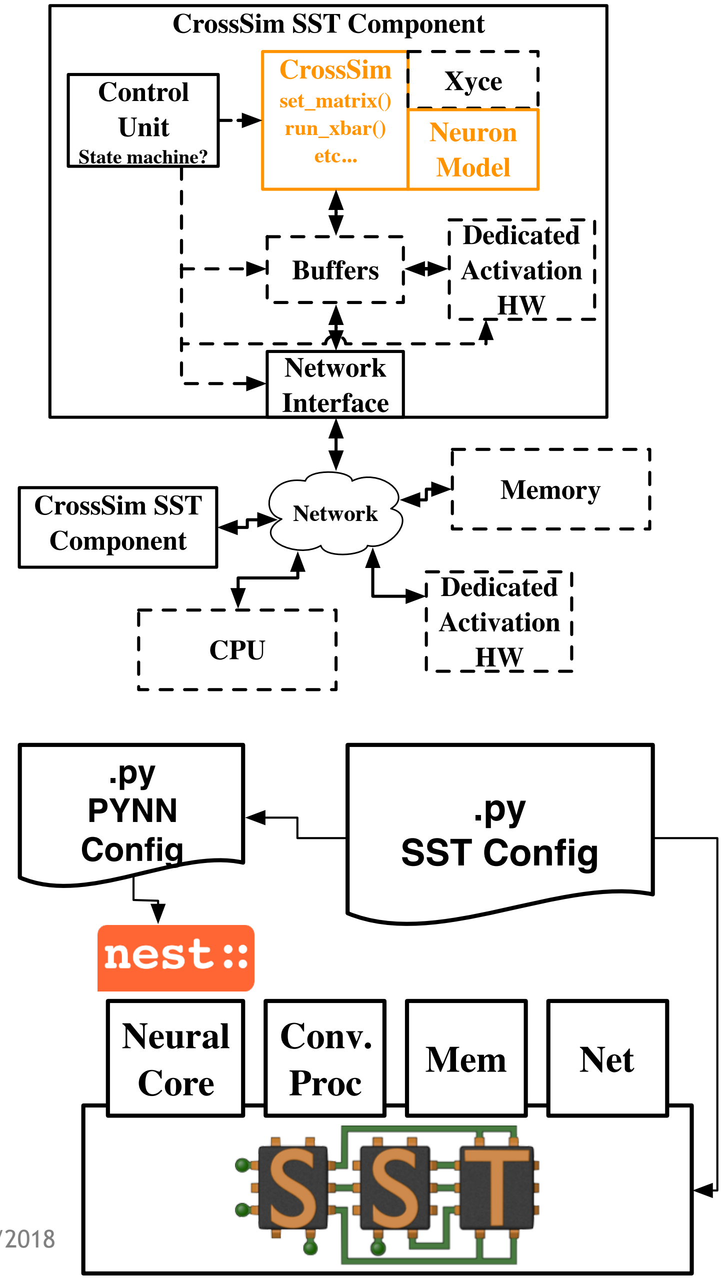
- RISC V
- Juno



Improved NoC Models

- Faster Performance
- NoC QoS
- Optical Circuit Routing

Future Directions: Neural Inspired



Custom processors / accelerators

Generic models (e.g. NEST, N2A)

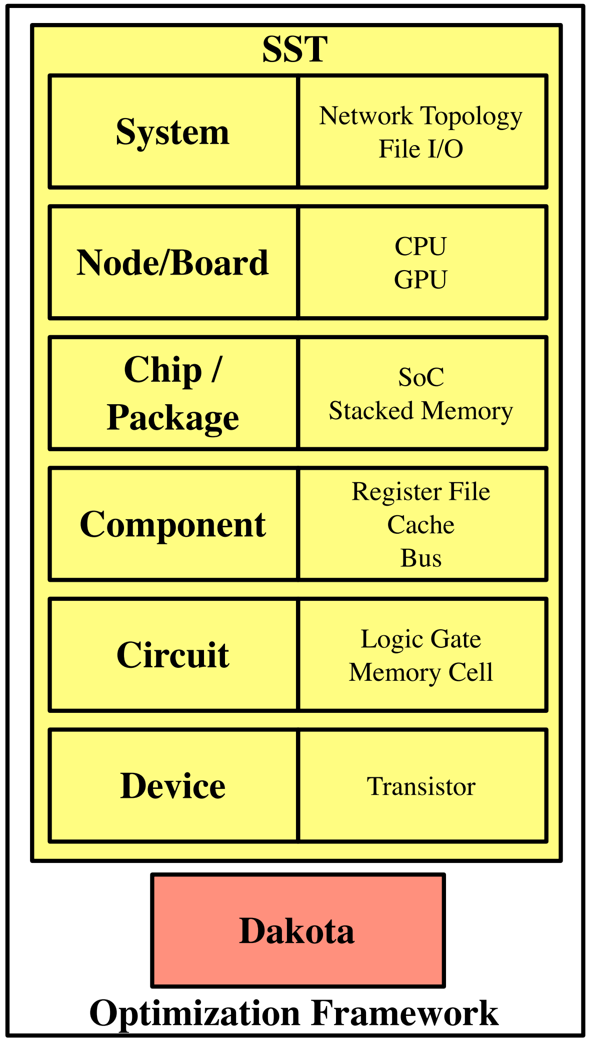
Explore

- System Integration Issues
- Architectural Bottlenecks
- Programmability

Allows…

- Exploration of conventional / Neural interface (e.g. different message routing protocols)
- Allows Neural "cores" to connect to conventional processors, network, memory, etc… (explore 'speeds and feeds')
- Scalability: SST can utilize thread- and MPI-level parallelism

Future Directions: More Framework Integration



Beyond Moore Computing

- New Architectures (e.g. Neuromorphic)
- New Devices (e.g. Memristor)
- New Programming Models / Algorithms

Requires Cross-Stack Optimization

- Device to System Level
- Use of Dakota / INDRA to automate design space exploration

Requires Inter-disciplinary approach

- SST Simulator as "Clearinghouse of Ideas"
- Common language of exploration

Conclusion

SST is a Parallel, Flexible, Open architectural simulator

Large library of Memory, Processor, Network, and other models

7.1 Release
- Usability enhancements in the Core
- New Memory, Network, Processor models

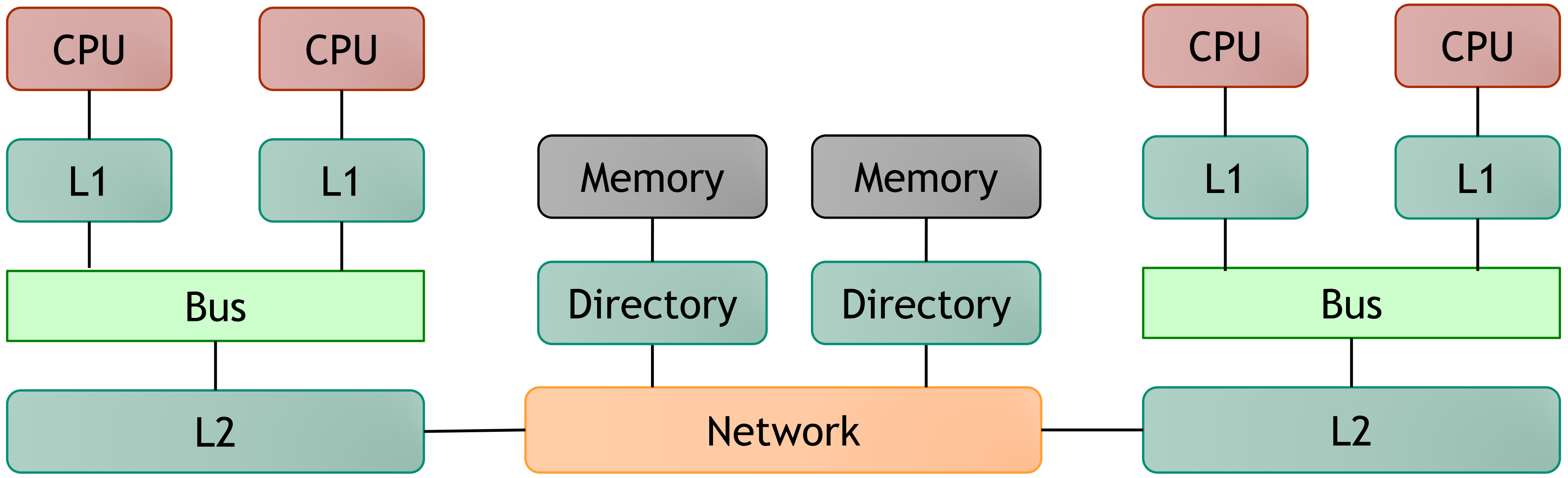
Future SST
- More & better components (RISC-V, Network, etc…)
- Chisel, Occam integration
- Beyond Moore Framework
- <Your Use Case Here>

Configuring a Simulation

Configuring a Simulation

SST uses a Python configuration file

- Defines global parameters for the simulation
- Defines and configures components
- Specifies links and link latencies between components



Part 1: Configure SST

Global simulation parameters

sst.setProgramOption("stopAtCycle", "100ms")

- Kill simulation (nicely!) if it runs to 100ms

sst.setProgramOption("timebase", "1ns")

- Tell SST that we're simulating at a granularity around 1ns
- Used by SST core when time units are not specified by a component
- Not a lower limit! (clocks can be > 1 GHz)

Part 2: Define components

Define: sst.Component("name", "type")

Configure: addParams ({ "parameter" : value, … })

Component name

Component type

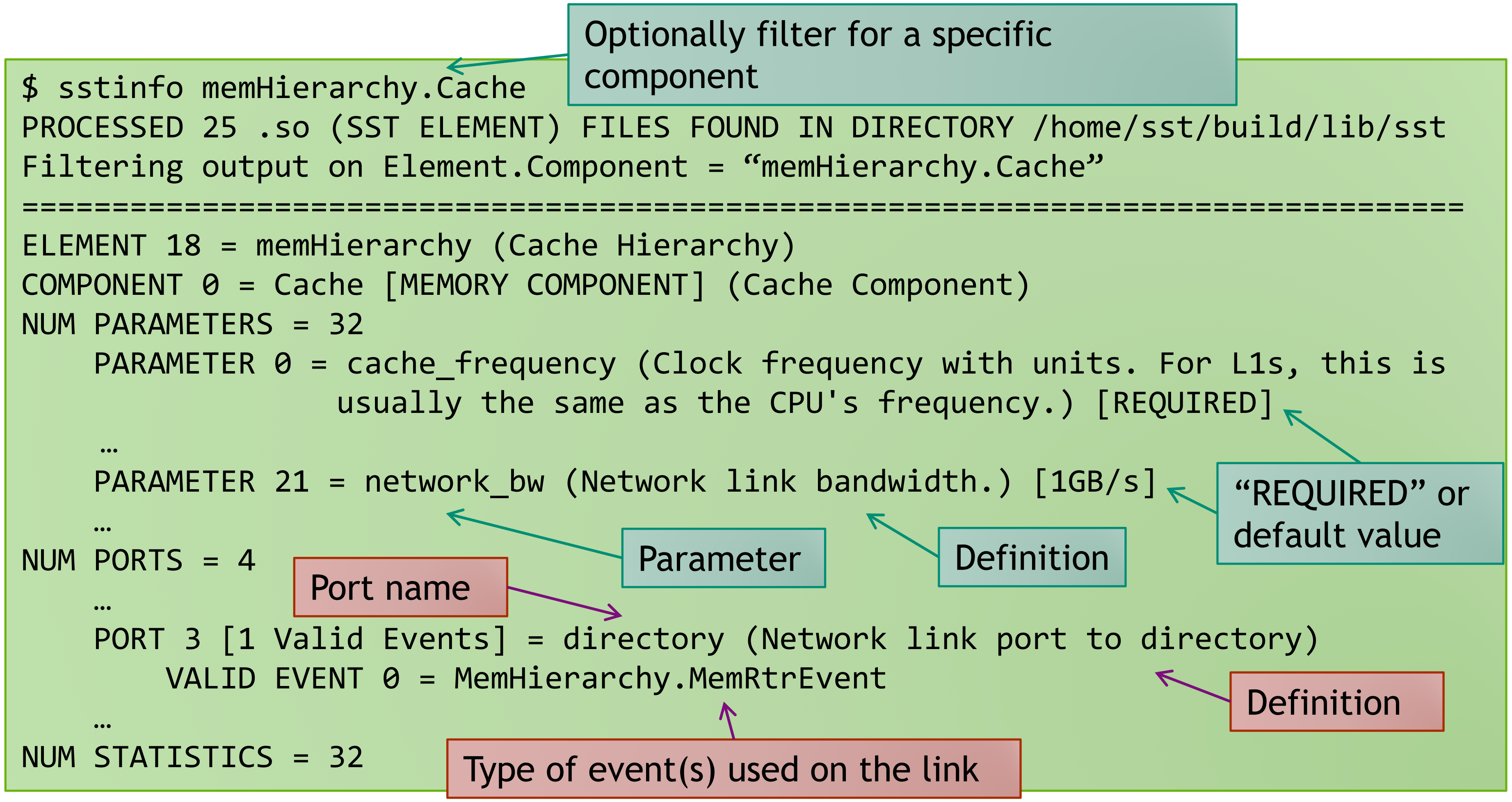
demo.py: line 172

```
network = sst.Component("router", "merlin.hr_router")
network.addParams({
    "xbar_bw" : "51.2GB/s",
    "link_bw" : "25.6GB/s",
    "num_ports" : 4,
    "flit_size" : "72B:
    "topology" : "merlin.singlerouter",
    "id" : "0",
    "input_buf_size" : "2KB",
    "output_buf_size" : "2KB"
})
```

Parameters

SSTInfo: Getting component info

Prints parameters, port names, and statistics


Optionally filter for a specific component
"REQUIRED" or default value
Parameter
Definition
Port name
Definition
Type of event(s) used on the link

```
$ sstinfo memHierarchy.Cache
PROCESSED 25 .so (SST ELEMENT) FILES FOUND IN DIRECTORY /home/sst/build/lib/sst
Filtering output on Element.Component = "memHierarchy.Cache"
================================================================================
ELEMENT 18 = memHierarchy (Cache Hierarchy)
COMPONENT 0 = Cache [MEMORY COMPONENT] (Cache Component)
NUM PARAMETERS = 32
    PARAMETER 0 = cache_frequency (Clock frequency with units. For L1s, this is
                  usually the same as the CPU's frequency.) [REQUIRED]
    …
    PARAMETER 21 = network_bw (Network link bandwidth.) [1GB/s]
    …
NUM PORTS = 4
    …
    PORT 3 [1 Valid Events] = directory (Network link port to directory)
        VALID EVENT 0 = MemHierarchy.MemRtrEvent
    …
NUM STATISTICS = 32
```

Part 3: Defining links

Example: Connect socket 0's L2 cache (l2cache0) to network

- Create a link: sst.Link("name")
- Define link endpoints: connect(endpoint1, endpoint2)
  - Endpoint is defined as: (Component, Port, Latency)
  - Note: Latencies of the two endpoints can differ

demo.py: line 220

Link name

```
…
l2cache0_network_link = sst.Link("l2cache0_network_link")
…
l2cache0_network_link.connect(
    (l2cache0, "directory", "50ps"),
    (network, "port0", "50ps") )
…
```

Endpoints

Running SST

Usage: sst [options] configFile.py

Common options:

`-v | --verbose`	Print verbose information during runtime
`--debug-file <filename>`	Send debugging output to specified file (default: sst_output)
`--add-lib-path <dirname>`	Add <dirname> to search path for element libraries
`--heartbeat-period <period>`	Every <period> time, print a heartbeat message
`--paritioner <zoltan | self | simple | rrobin | linear | lib.partitioner.name>`	Specify the partitioning mechanism for parallel runs
`--model-options “<args>”`	Command line arguments to send to the Python configuration file
`--output-partition <filename>`	Write partitioning information to <filename>
`--output-dot <filename>`	Output a graph representing the configuration in “Dot” format to <filename>

Running a Simulation

Launch simulation

```
$ sst demo.py
```

Output

```
Inserting stop event at cycle 100ms, 100000000000
ARIEL-SST PIN tool activating with 4 threads
ARIEL: Default memory pool set to 0
ARIEL: Tool is configured to begin with profiling immediately.
ARIEL: Starting program.
Performing iteration 0
Performing iteration 0
Performing iteration 0
Performing iteration 0
…
…
Simulation is complete, simulated time: 125.209 us
```

Finally: Getting help

SST wiki contains lots of information (www.sst-simulator.org)

- Downloading, installing, and running SST
- Element libraries and external components
- Guides for extending SST
- Information on APIs
- Information about current development efforts

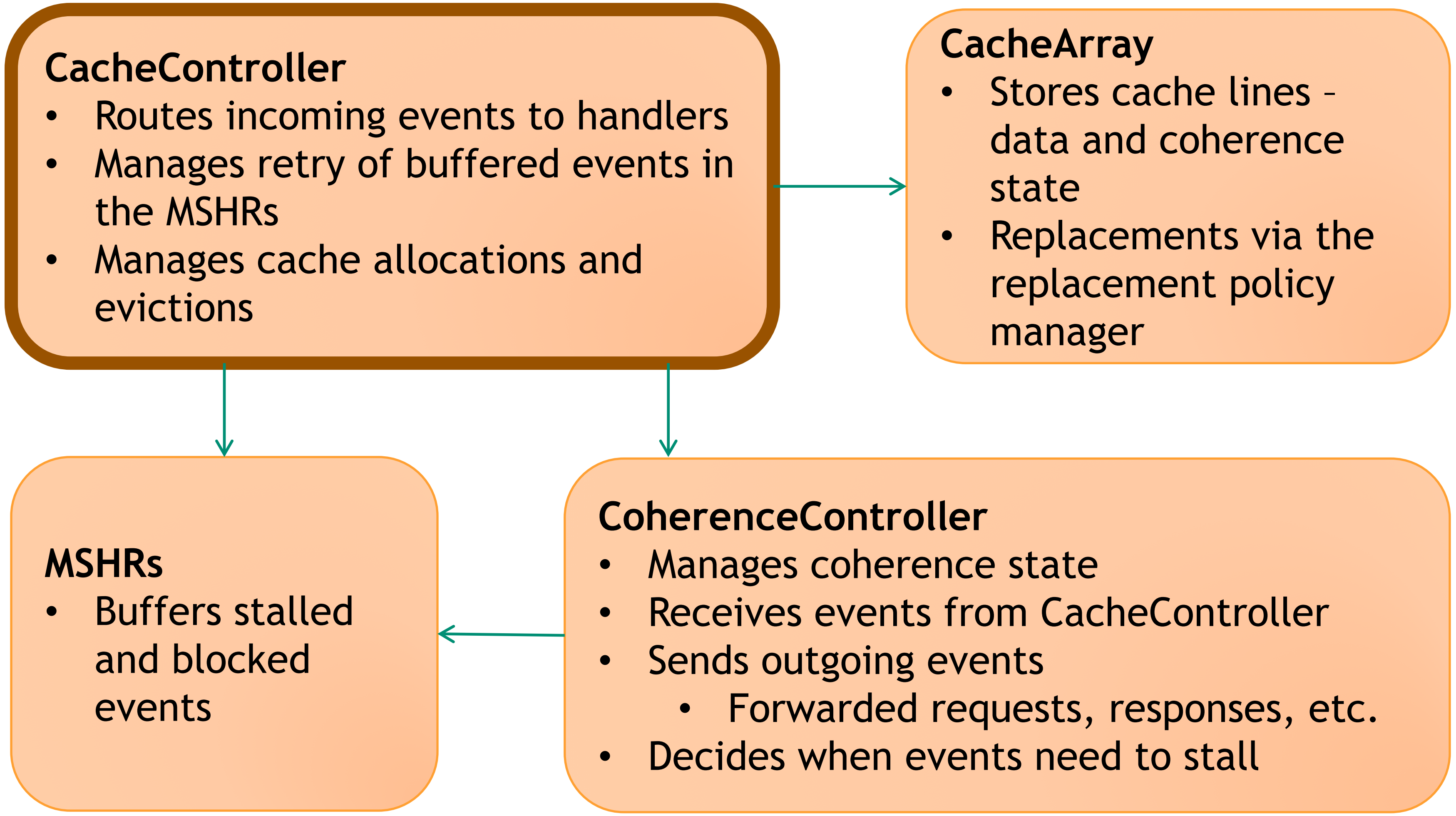
https://github.com/sstsimulator

Getting and installing SST

www.sst-simulator.org

- Current release (7.1) source download
    - https://github.com/sstsimulator/sst-elements
- Detailed build instructions including dependencies for Linux & Mac
- Links to mailing lists for updates and support

MemHierarchy: Cache structure



CacheController
- Routes incoming events to handlers
- Manages retry of buffered events in the MSHRs
- Manages cache allocations and evictions

CacheArray
- Stores cache lines – data and coherence state
- Replacements via the replacement policy manager

MSHRs
- Buffers stalled and blocked events

CoherenceController
- Manages coherence state
- Receives events from CacheController
- Sends outgoing events
  - Forwarded requests, responses, etc.
- Decides when events need to stall

MemHierarchy: Main memory

MemoryController

- Contains a 'backing store' for simulated data
- Can communicate over a network or via a direct link with a cache or directory
- Interfaces with multiple memory backends

Available backends

- SimpleMem – basic read/write with associated latencies
- DRAMSim2 – DRAM (external)
- NVDIMMSim – Non-volatile memory (e.g., Flash) (external)
- HybridSim – non-volatile memory with a DRAM cache (external)
- VaultSimC – stacked DRAM

Merlin: Network simulator

Low-level, flexible networking components that can be used to simulate high-speed networks (machine level) or on-chip networks

Capabilities

- High radix router model (hr_router)
- Topologies – mesh, n-dim tori, fat-tree, dragonfly

Many ways to drive a network

- Simple traffic generation models
  - Nearest neighbor, uniform, uniform w/ hotspot, normal, binomial
- MemHierarchy
- Lightweight network endpoint models (Ember – coming up next)
- Or, make your own

Ember: Network traffic generator

Light-weight endpoint for modeling network traffic

- Enables large-scale simulation of networks where detailed modeling of endpoints would be expensive

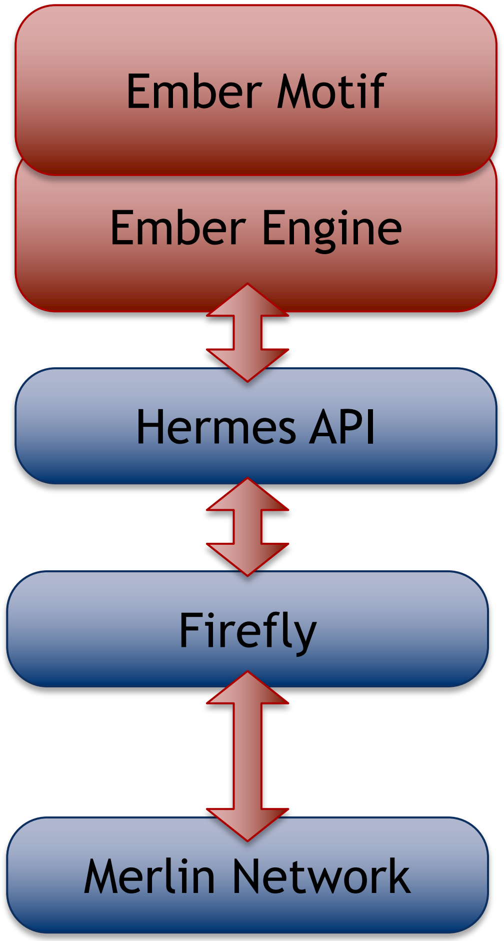
Packages patterns as motifs

- Can encode a high level of complexity in the patterns
- Generic method for users to extend SST with additional communication patterns

Intended to be a driver for the Hermes, Firefly, and Merlin communication modeling stack

- Uses Hermes message API to create communications
- Abstracted from low-level, allowing modular reuse of additional hardware models

Ember: Overview



High Level Communication Pattern and Logic
Generates communication events

Event to Message Call, Motif Management
Handles the tracking of the motif

Message Passing Semantics
Collectives, Matching etc

Packetization and Byte Movement Engine
Generates packets and coordinates with network

Flit Level Movement, Routing, Delivery
Moves flits across network, timing etc

Ember: Motifs

Motifs are lightweight patterns of communication

- Tend to have very small state
- Extracted from parent applications
- Models as an MPI program (serial flow of control)
  - Many motifs acting in the simulation create the parallel behavior

Example motifs

- Halo exchanges (1, 2, and 3D)
- MPI collections – reductions, all-reduce, gather, barrier
- Communication sweeping (Sweep3D, LU, etc.)

Ember: Motifs (continued)

The EmberEngine creates and manages the motif

- Creates an event queue which the motif adds events to when probed
- The Engine executes the queued events in order, converting them to message semantic calls as needed
- When the queue is empty, the motif is probed again for events

Events correspond to a specific action

- E.g., send, recv, allreduce, compute-for-a-period, wait, etc.

Firefly: Network traffic

Purpose: Create network traffic, based on application communication patterns, at large scale

- Enables testing the impact of network topologies and technologies on application communication at very large scale

Scales to 1 million nodes

Supports multiple "cores" per Node

- Interaction between cores limited to message passing

Supports space sharing of the network

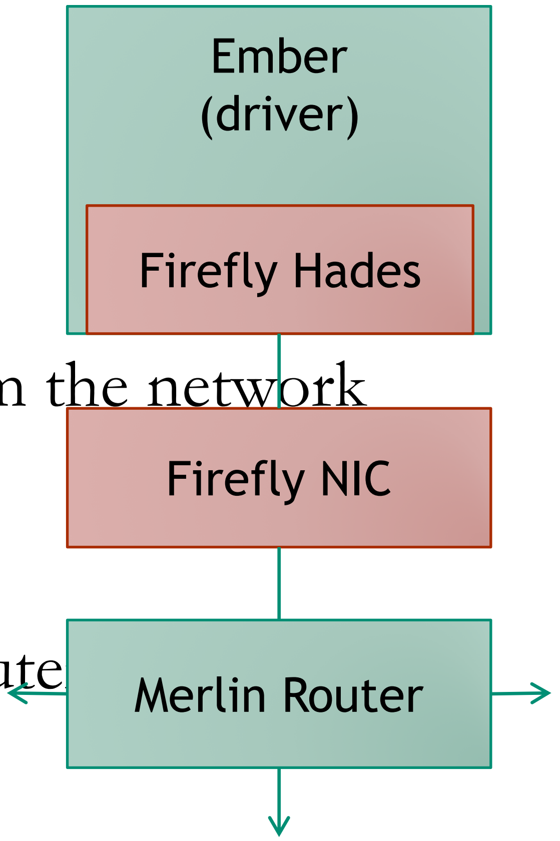
- Multiple "apps" running simultaneously

Firefly: Simulating large networks

A network node consists of
- Driver (the "application")
- NIC
- Router

Nodes are connected together via the routers to form the network
- Fat tree, torus, etc.

Firefly is the interface between the driver and the route
- Message passing library → Firefly Hades
- NIC → Firefly NIC



Scheduler

Models HPC system-wide job scheduling

Three components

- Sched: schedules and allocates resources for a stream of jobs
- Node: runs scheduled jobs on their allocated resources
- FaultInjection: injects failures onto the resources

The scheduler is currently a stand-alone element library

- The schedComponent and nodeComponent must be used together
- The faultInjectionComponent is optional